Instruktioner för datortentamen Torsdag 2019-01-17 (14–19)

TDDD73 Funktionell och imperativ programmering i Python
TDDE24 Funktionell och imperativ programmering del 2

Hjälpmedel

Följande hjälpmedel är tillåtna:

- Exakt en valfri bok, t.ex. den rekommenderade kursboken. Boken får ha anteckningar, men inga lösa lappar som inte hör till bokens innehåll.
- Exakt ett A4-papper med egna anteckningar om precis vad som helst. Det kan vara hand- eller maskinskrivet, på ena eller båda sidorna.
- Pennor, radergummin och liknande för att kunna skissa lösningar på papper.

Icke tillåtna hjälpmedel inkluderar bl.a. alla former av elektronisk utrustning samt böcker och anteckningar utöver det som listats ovan.

Lösa tomma papper för att göra anteckningar tillhandahålls av tentamensvakterna.

Genomförande

Under datortentan kommer du att arbeta i en begränsad och övervakad datormiljö. Med hjälp av en särskild **tentaklient** skickar du in dina svar. Varje uppgift kan **skickas in en enda gång** och kommer därefter att poängsättas. Man kan alltså inte komplettera.

Poängsättning kan *i viss mån* ske redan under tentans gång, men detta är inte garanterat utan beror bland annat på antalet tillgängliga rättare.

Du kan även använda tentaklienten för att **ställa frågor** till jourhavande, som vid behov kan vidarebefordra till examinator. Besök i skrivsalen kommer endast att ske vid allvarligare tekniska problem.

För att underlätta rättningen:

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py

Under en datortenta har man unika möjligheter att förtydliga uppgifter för *alla* när en person ställer en fråga om hur en uppgift ska tolkas. I den här versionen av tentan har vi lagt in dessa förtydliganden i rutor av den här typen. Övriga delar av dokumentet är kvar i ursprunglig form.

Betygsättning

Datortentan består av totalt 6 uppgifter som vardera kan ge maximalt 5 poäng. Alla uppgifter är utvalda för att testa olika aspekter av kursinnehållet. De är också tänkta att ha olika svårighetsgrad och är ordnade så att den svårigheten oftast ökar mot slutet. Observera att vad som är lätt eller svårt naturligtvis skiljer sig från person till person, så det är alltid en bra idé att snabbt läsa igenom alla uppgifterna för att bättre kunna prioritera arbetet.

Betygssättningen sker enligt följande tabell:

- För betyg 3 krävs minst 12 poäng.
- För betyg 4 krävs minst 19 poäng.
- För betyg 5 krävs minst 25 poäng.

Rättningskriterier

Följande allmänna kriterier används vid poängsättning av uppgifter. Lösningar som bryter mot dessa kommer att få poängavdrag.

- Funktioner ska ha **exakt samma namn** som i uppgiften. Detta underlättar rättningen.
- **Namn** på parametrar, variabler och hjälpfunktioner som inte specificerats explicit i uppgiften ska vara **beskrivande** och följa **namnstandarden**.
- Lösningen ska vara **välstrukturerad och väldokumenterad** på samma sätt som under kursens laborationer. Detta inkluderar att skriva **docstrings för samtliga funktioner**, även hjälpfunktioner som man själv inför. Specialtaggar som `:param:` krävs ej.

> Förtydligande skickat 14:28: Det går bra att skriva docstrings på svenska eller engelska. Docstrings behövs inte för lambda-uttryck, bara för namngivna funktioner. I uppgift 4b skapar ni en funktion i ett enda uttryck, istället för att använda def. Där behövs heller inte någon docstring.

- Lösningen ska följa de **specifika regler och villkor** som uppgiften har satt upp. Det kan t.ex. röra sig om att vissa funktioner eller metoder ej får användas, eller att lösningen ska göras enligt en särskild modell.
- Lösningen ska fungera **exakt som i körexemplen** i uppgiften, om inte texten indikerar något annat.
- Lösningen ska vara **generell**, d.v.s. den ska inte enbart fungera för de körexempel som finns i uppgiften, utan för alla tänkbara indata som följer uppställningen i frågan.

  Man behöver dock normalt **inte felkontrollera** indata, utom när uppgiften särskilt anger det. Funktioner måste alltså ge korrekta svar för korrekta indata enligt uppgiften, men får normalt krascha eller ge felaktiga svar för felaktiga indata (såsom när en funktion som ska hantera heltal ges en sträng som parameter).

- Den implementerade lösningen ska kunna köras inom **rimliga tidsramar**. Till exempel accepteras inte en lösning som tar 1 minut för att konkatenera två strängar som är 4 bokstäver långa. Eventuella undantag anges uttryckligen i frågan.

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py

Uppgift 1

Vi är intresserade av att hitta det ***minsta strikt positiva talet*** och det ***största negativa talet*** i en rak lista av heltal. Din uppgift blir därför att skriva funktionen `minpos_maxneg(seq)`, som returnerar en tupel innehållande dessa två värden i den angivna ordningen:

- `assert minpos_maxneg([0, 5, 2, -1, -50, 2, -15, -2, 55, 56]) == (2, -1)`

Kopiera gärna in i dessa tester i din egen kodfil för att förenkla din egen testning!

Om listan saknar strikt positiva tal, ska det första värdet i den returnerade tupeln vara `None`. Om listan saknar negativa tal, ska det andra värdet i den returnerade tupeln vara `None`. Nedan följer ett exempel där listan varken har positiva eller negativa tal.

- `assert minpos_maxneg([]) == (None, None)`

Ytterligare villkor:

- Lösningen får vara iterativ eller rekursiv. Det är också tillåtet att använda både listbyggare (list comprehensions) och inbyggda funktioner som arbetar på hela listor. Dock måste man fortfarande ha ett sätt att hantera fallet när listan saknar positiva och/eller negativa tal.
- Funktionen ska vara funktionell i bemärkelsen att inskickade parametrar inte får modifieras.

Bra att tänka på

- Alla funktioner ska dokumenteras med docstrings. Detta är sista varningen!
- De testfall vi ger i tentan är bara exempel, och är främst till för att förtydliga vad vi menar med beskrivningstexten. Det finns inga garantier för att en lösning som uppfyller de här testfallen är helt korrekt. Skapa gärna egna tester som kan täcka ytterligare fall.

  Testfall *får* men *behöver inte* lämnas in.
- Testa efter varje ändring, även när du *tror* att ändringen var harmlös. Inlämnad kod måste vara körbar.

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py

Uppgift 2

Skriv en funktion `sum_pairs(seq)` som tar en sekvens av heltal och returnerar:

- Tomma listan om `seq` är tom
- Den ursprungliga sekvensen om `seq` har 1 element
- Annars, listan av alla "par-summor" av två element som följer direkt efter varandra i `seq`.

Exempel:

- `sum_pairs([1,2,5,12,27])` blir `[3, 7, 17, 39]`, det vill säga `1+2`, `2+5`, `5+12` och `12+27`.

Funktionen ska finnas i **två varianter** som ska fungera "exakt likadant", bortsett från att de ska ha olika lösningsmodeller:

- `sum_pairs_r` ska arbeta enligt en rekursiv modell.
- `sum_pairs_i` ska arbeta enligt en iterativ modell.

Ytterligare villkor:

- Du får inte använda inbyggda funktioner som behandlar alla element i hela listor. Däremot är givetvis `len()` tillåten.
- Du får inte använda listbyggare (eng. list comprehensions).
- Lösningarna ska vara funktionella i bemärkelsen att de inte får modifiera listan som skickas in.

Funktionerna ska bland annat klara följande **tester**.

- `assert sum_pairs_i(()) == []`
- `assert sum_pairs_i([1, 2, 5, 12, 27]) == [3, 7, 17, 39]`
- `assert sum_pairs_i([1, 2, 5, -32, 8, 539, 2, 53, 1]) == [3, 7, -27, -24, 547, 541, 55, 54]`
- `assert sum_pairs_r(()) == []`
- `assert sum_pairs_r([1, 2, 5, 12, 27]) == [3, 7, 17, 39]`
- `assert sum_pairs_r([1, 2, 5, -32, 8, 539, 2, 53, 1]) == [3, 7, -27, -24, 547, 541, 55, 54]`

De båda funktionerna ger vardera 2.5 poäng, och det är möjligt att få poäng på dem oberoende av varandra.

Uppgift 3

Skriv en funktion `power_of_each(lst)`, som:

- Tar en lista som kan innehålla godtyckligt nästlade tomma och icke-tomma listor, där alla element som inte är listor är tal
- Returnerar en lista med samma nästlade struktur, men där man för varje element *r* som *inte* är en lista istället har med talet 2^r (2 upphöjt till r).

Tänk på att eftersom listan är nästlad, kan även elementen vara godtyckliga nästlade listor.

Ytterligare villkor:

- Funktionen ska vara funktionell i bemärkelsen att inskickade parametrar inte får modifieras.
- Det är tillåtet att lösa uppgiften rekursivt eller iterativt, eller med en kombination av dessa metoder.
- Det är tillåtet att använda inbyggda funktioner som arbetar på hela listor. Det är också tillåtet att använda listbyggare (list comprehensions), men dessa arbetar inte själva på nästlade listor.

Exempel:

- `assert power_of_each([3, 2, 3, 16]) == [8, 4, 8, 65536]`
- `assert power_of_each([-3]) == [0.125]`
- `assert power_of_each([0, 1, [2, [3, [[[4]], 3]]]]) == [1, 2, [4, [8, [[[16]], 8]]]]`

Uppgift 4

Deluppgift 4a (3p)

Skapa en högre ordningens funktion `multiple_apply(f, n)`. Funktionen ska **ta** en funktion `f` och ett icke-negativt heltal `n` som argument och **returnera** en ny funktion, som i sin tur:

- Tar ett tal som argument
- Returnerar `f(f(f(...(x))))`, där `f` appliceras totalt `n` gånger.

Exempel:

- Anta att vi sätter `newfun = multiple_apply(f, 3)`, där `f` är någon godtycklig funktion som tar exakt 1 parameter.
- Då ska `newfun(10)` returnera `f(f(f(10)))`.

Ytterligare villkor:

- Din lösning ska klara att `n==0`. Exempel ges nedan.
- Funktionen ska vara funktionell i bemärkelsen att inskickade parametrar inte får modifieras.
- Det är tillåtet att använda inbyggda funktioner som arbetar på hela listor. Det är också tillåtet att använda listbyggare (list comprehensions).

Förtydligande skickat 16:43: I 4a hänvisas till ett exempel ”nedan”. Här menas ett av exemplen som ges under 4b, där man ser hur n==0 hanteras för ett konkret fall.

Förtydligande skickat 16:46: I 4a:
- Tar ett tal som argument
kan förtydligas till
- tar ett tal x som argument

Deluppgift 4b (2p)

Använd `multiple_apply` för att definiera `pow2mult(n, c)`, som returnerar $2^n \cdot c$ givet att $n >= 0$. Detta ska definieras med hjälp av ett enda uttryck:

- `pow2mult = ...`

Du ska göra detta utan att definiera egna hjälpfunktioner utöver `multiple_apply`.

Exempel:

- `assert pow2mult(3, 1) == 8`
- `assert pow2mult(3, 3) == 24`

- `assert pow2mult(0, 3) == 3`

Uppgift 5

Anta att vi har två godtyckliga strängar, t.ex. `part1="abc"` och `part2="def"`. Då kan vi bilda många nya strängar genom att blanda tecknen från strängarna men ändå bevara den ursprungliga ordningen inom varje sträng. Till exempel kan vi bilda `"abcdef"`, men även `"adbecf"`, som också har både *a-före-b-före-c* och *d-före-e-före-f*. Vi kan däremot inte skapa `"abedcf"`, eftersom vi där har placerat *e* före *d*.

Skriv en funktion `can_interleave(part1, part2, whole)`, som tar två sådana strängar, `part1` och `part2`, och testar om den nya strängen `whole` kan skapas från dessa. Funktionen ska returnera `True` om detta är möjligt och `False` annars.

Ytterligare villkor:

- Funktionen behöver *inte* vara minnes- eller tidseffektiv. Det är till exempel acceptabelt att lösningen provar alla alternativ.
- Lösningen får vara iterativ eller rekursiv. Det är också tillåtet att använda både list-byggare (list comprehensions) och inbyggda funktioner som arbetar på hela listor.
- Funktionen ska vara funktionell i bemärkelsen att parametrar inte får ändras.

Exempel:

- `assert can_interleave("abc", "def", "abcde") == False`
- `assert can_interleave("abc", "def", "abcdef") == True`
- `assert can_interleave("abc", "def", "acbedf") == False`

Förtydligande skickat 15:33: I fråga 5 ska vi blanda tecknen från två strängar. När detta är gjort har vi kvar lika många tecken i whole som vi hade i de två ursprungssträngarna, utan att något har försvunnit. Det står att strängarna är godtyckliga. Det betyder att inga kombinationer av tecken eller strängar är omöjliga som input.

Uppgift 6

Look-and-say-sekvensen, som introducerades av John Conway, är en sekvens av heltal som ser ut som följer:

$$1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, \ldots$$

Detta skapas genom att man börjar med talet $a_1 = 1$ och därefter, för varje efterkommande tal a_{i+1}, "läser upp" antalet siffror av en viss typ i a_i:

- Vi har "1", som består av "en etta": 1 1 ger 11
- Vi har "11", som består av "två ettor": 2 1 ger 21
- Vi har "21", som består av "en tvåa, en etta": 1 2, 1 1 ger 1211
- Vi har "1211", som består av "en etta, en tvåa, två ettor": 1 1, 1 2, 2 1 ger 111221
- ...

Man kan enkelt generalisera detta genom att börja med andra heltal, t.ex. $a_1 = 42$, men applicera samma regel för att få nästa tal i sekvensen.

Att göra

Skriv en funktion `las(a1, n)` som returnerar en lista som innehåller de första `n` talen i look-and-say-sekvensen, med start i $a_i =$ `a1`. Man får anta att `n` ≥ 0. Exempel:

- `assert las(42, 0) == []`
- `assert las(1, 5) == [1, 11, 21, 1211, 111221]`
- `assert las(42, 5) == [42, 1412, 11141112, 31143112, 132114132112]`

Tips: Det är lättare att manipulera enskilda siffror i ett tal om det (ibland) representeras som en sträng.

Förtydligande skickat 16:17: I uppgift 6 får man anta att a1 >= 0. Ni behöver aldrig hantera t.ex. ”1 minustecken”.